

# EXHIBIT V

**MICROCOMPUTERS**  
**698 Chapter 10**

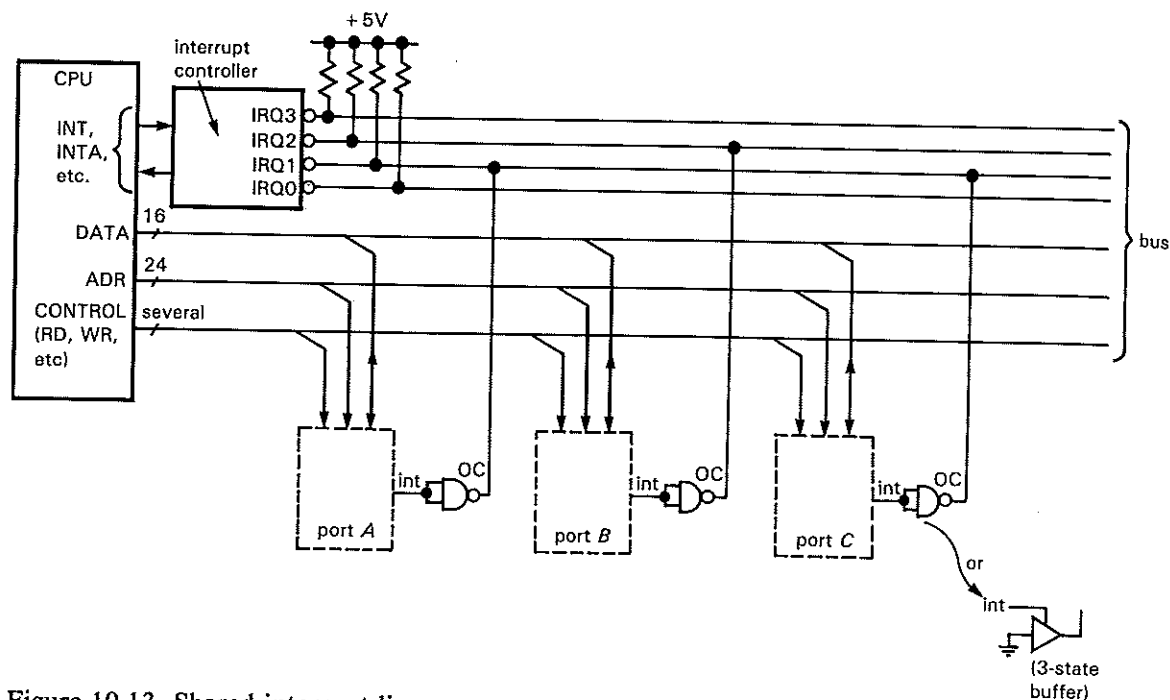


Figure 10.13. Shared interrupt lines.

□ **Shared interrupt lines**

The usual interrupt protocol, as implemented on many microcomputers, circumvents these limitations. Look at Figure 10.13. There are several (prioritized) IRQ-type lines; these are negative-true inputs to the CPU (or its immediate support circuitry). To request an interrupt, you pull one of the IRQ' lines LOW, using an open-collector (or three-state) gate, as shown (note the trick for using a three-state gate to mimic an open-collector gate). The IRQ' lines are shared, with a single resistive pullup, so you can put as many devices on each IRQ' line as you want; in our example two ports share IRQ1. You would generally connect a latency-sensitive (impatient) device to a higher-priority IRQ' line.

Since the IRQ' lines are shared, there could always be another device interrupting on the same line at the same time. The CPU needs to know who interrupted, so it can jump to the appropriate handler. There is a simple way, and a complicated

way, to do this. The simple way is called *autovector polling* and is used nearly universally (though not on the IBM PC). Here's how it works.

□ **Autovector polling.** Some circuitry on the CPU board (we'll have an example in Chapter 11) instructs the microprocessor that it is to use autovectoring, which works just like the IBM PC – each level of interrupt forces a jump through a corresponding vectoring location in low memory. For example, the 68000 microprocessor family we'll meet in Chapter 11 has seven levels of prioritized interrupt, which autovector through 4-byte pointers stored in the 28 ( $7 \times 4$ ) locations  $64_H$  through  $7F_H$ . You put the addresses of the handlers in those locations, just as in our example above. For example, you would put the (4-byte) address of the handler for a level-3 interrupt in hex locations  $6C$  through  $6F$ .

Once in the handler, you know which level of interrupt you're servicing; you just don't know which particular device caused the interrupt. To find out, you

simply check the status registers of each of the devices connected to that level of interrupt (a device *never* requests an interrupt without also indicating its need by setting one or more readable status bits). If a bit is set indicating that something needs to be done, you do it, including whatever it takes to cause the device to disassert its IRQ'. Some devices (like our keyboard) clear their interrupt when read, whereas others may need a particular byte sent to some I/O port address.

If the device you serviced was the only one interrupting at that level, that IRQ' will now be HIGH upon returning to the interrupted program, and execution will continue. However, if there had been a second interrupting device at the same level, that IRQ' line will still be held LOW (by the wired-OR action of the shared IRQ' line) upon return from the service routine, so the CPU will immediately autovector back to the same handler. This time the polling will find the other interrupting device, do its thing, and return. Note that the order in which you poll status registers effectively sets up a "software priority," in addition to the hardware priority of the multiple IRQ' levels.

- *Interrupt acknowledgment.* We shouldn't leave the subject of interrupts without mentioning a more sophisticated procedure for identifying who interrupted – *interrupt acknowledgment*. In this method the CPU doesn't need to poll the status registers of possible interrupters, because the interrupting device *tells* the CPU its name, when asked. The interrupter does this by putting an "interrupt vector" (usually a unique 8-bit quantity) onto the DATA lines in response to an "interrupt acknowledge" signal that the CPU generates during the interrupt processing.

Nearly every microprocessor generates the needed signals. The sequence of events

goes like this: (a) The CPU notices a pending interrupt. (b) The CPU finishes the current instruction, then asserts (i) bus signals that announce an interrupt, (ii) the interrupt level being serviced (on the low-order ADDRESS lines), and (iii) READ-like strobes that invite the interrupting device to identify itself. (c) The interrupting device responds to this bus activity by asserting its identity (interrupt vector) onto the DATA lines. (d) The CPU reads the vector and jumps into the corresponding unique handler for the interrupting device. (e) The handler software, as in our last example, reads flags, gets and sends data, etc., as needed; among its other duties, it must make sure the interrupting device disasserts its interrupt. (f) Finally, the interrupt handler software returns control to the program that was interrupted.

Sharp-eyed readers may have noticed a flaw in the procedure just outlined. In particular, there has to be a protocol to ensure that only one device asserts its vector, since there may be several simultaneous interrupting devices at the same IRQ level. The usual way to handle this is to have a bus signal (call it INTP, "interrupt priority") that is unusual in not being shared by devices on the bus, but rather is passed along *through* each device's interface circuit, beginning as a HIGH level at the device closest to the CPU and threading along through each interface. That's called a "daisy chain" in the colorful language of electronics. The rule for INTP hardware logic is as follows: If you have not requested an interrupt at the level being acknowledged, pass INTP through to the next device unchanged; if you *have* interrupted at that level, hold your INTP output LOW. Now the rule for asserting your vector goes like this: Put your vector number onto the data bus when requested by the CPU only if (a) you have an interrupt pending at the level being acknowledged and (b) your input INTP is HIGH. This guarantees that

## MICROCOMPUTERS

## 700 Chapter 10

only one device asserts its vector; it also establishes a "serial priority" chain within each IRQ level, with devices electrically closest to the CPU getting serviced first. Computers that implement this scheme have little jumper plugs to pass INTP over unused motherboard slots. Don't forget to remove these jumpers when you plug in a new interface card (and put them back when you take one out!).

There is a nice alternative to the serial daisy-chain method of interrupt acknowledgment: Instead of threading a line through each possible interrupter, you bring individual lines back from each one to a priority encoder (Section 8.14), which in turn acknowledges the interrupt by asserting the identity of the highest-priority interrupting device. This scheme avoids the nuisance of daisy-chain jumpers. We will describe it in detail in Section 11.4 (Fig. 11.8).

In most microcomputer systems it isn't worth implementing the full-blown interrupt acknowledgment just described. After all, with 8-level autovectoring you can handle up to 8 interrupting devices without polling, and several times that number with polling. Only in large computer systems, in which you demand fast response with dozens of interrupting devices present, might you succumb to the complexity of the interrupt acknowledgment protocol, whether with serial daisy-chained hardware priority or with parallel priority encoding.

However, it is important to realize that even simple computers may be using vectored interrupt acknowledgment *internally*. For example, the simple 6-level autovector interrupt scheme of the IBM PC seen by the bus user is actually generated by an 8259 "programmable interrupt controller" chip that lives close to the CPU and generates the proper interrupt acknowledgment sequence just described (see below). This is necessary because the 8086 (and successors) can't implement autovectoring

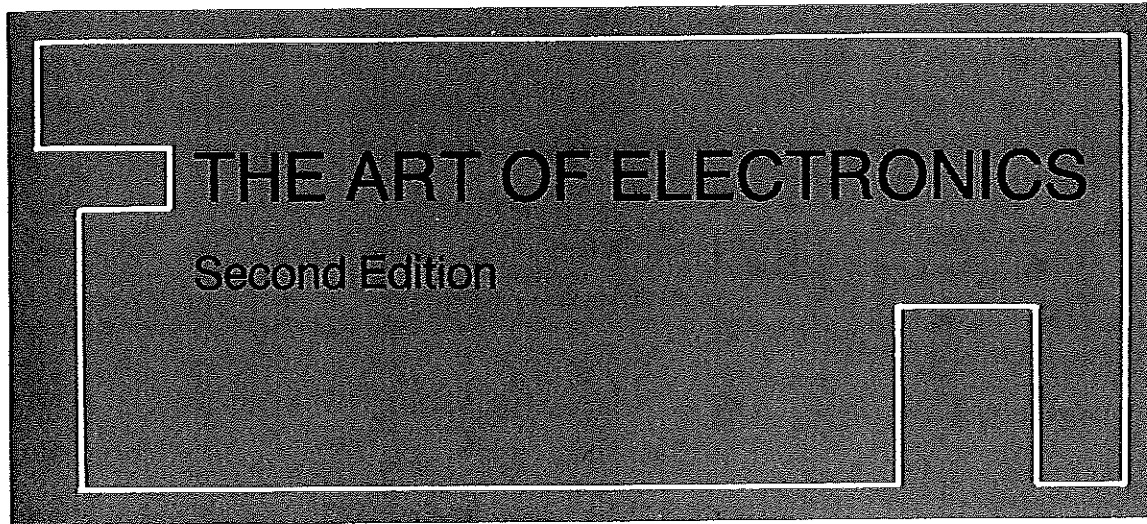
by themselves. On the other hand, the popular 68000 series of CPU chips can implement autovectoring internally, with just a single external gate (see Chapter 11).

#### □ Interrupt masks

We put a flip-flop in our simple keyboard example so that its interrupts could be disabled, even though the 8259 controller lets you turn off ("mask") each level of interrupt individually. We did that so that some other device could then use IRQ2. For a bus with shared (*level-sensitive*) IRQ' lines, it is especially important to make each interrupt source maskable, again with an I/O output port bit. For example, a printer port normally interrupts each time its output buffer is empty ("give me more data"); when you've finished printing, though, you don't care. The obvious solution is to turn off printer interrupts. Since there might be other devices hooked to the same interrupt level, you must not mask that whole level; instead, you just send a bit to the printer port to disable its interrupts.

#### □ How the IBM PC got the way it is

The 8086/8 microprocessor used in the IBM PC actually implements the full vectored interrupt acknowledgment protocol. To keep things simple, however, the PC designers used an 8259 interrupt controller IC on the motherboard. The way it is used in the PC, it has a set of IRQ inputs from the I/O bus card slots (that's where you make your interrupt requests), and it connects to the microprocessor's data bus and signal lines. When it gets a request on an IRQ line from a peripheral, it figures out priority and goes through the whole business of asserting the corresponding vector onto the data bus. It has a mask register (accessible as I/O port 21<sub>H</sub>) so that you can disable any specified group of interrupts.



**Paul Horowitz** HARVARD UNIVERSITY

**Winfield Hill** ROWLAND INSTITUTE FOR SCIENCE, CAMBRIDGE, MASSACHUSETTS



**CAMBRIDGE**  
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE  
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS

The Edinburgh Building, Cambridge CB2 2RU, UK <http://www.cup.cam.ac.uk>

40 West 20th Street, New York, NY 10011-4211, USA <http://www.cup.org>

10 Stamford Road, Oakleigh, Melbourne 3166, Australia

Ruiz de Alarcón 13, 28014 Madrid, Spain

© Cambridge University Press 1980, 1989

This book is in copyright. Subject to statutory exception  
and to the provisions of relevant collective licensing agreements,  
no reproduction of any part may take place without  
the written permission of Cambridge University Press.

First edition published 1980

Second edition published 1989

Reprinted 1990 (twice), 1991, 1993, 1994 (twice), 1995, 1996, 1997,  
1998 (twice), 1999

Printed in the United States of America

Typeset in Times

*A catalog record for this book is available from the British Library*

*Library of Congress Cataloging in Publication Data is available*

ISBN 0 521 37095 7 hardback